The language

Fundamentals
- Conventions
- Syntax

The stack
- Stack machine model
- Stack effect declarations
- Stack effect checking

Basic data types
- Booleans
- Numbers
- Collections

Evaluation
- Words
- Shuffle words
- Combinators
- Co-operative threads

Named values
- Lexical variables
- Dynamic variables
- Global variables

Abstractions
- Fried quotations
- Objects
- Exception handling
- Deterministic resource disposal
- Memoization
- Parsing words
- Macros
- Continuations

Program organization
- Vocabulary loader

Vocabularies tagged **extensions** implement various additional language abstractions.

Conventions

Various conventions are used throughout the Factor documentation and source code.

Glossary of terms

Common terminology and abbreviations used throughout Factor and its documentation:

Term	Definition
alist	an association list; see **Association lists**
assoc	an associative mapping; see **Associative mapping operations**
associative mapping	an object whose class implements the **Associative mapping protocol**
boolean	**t** or **f**
class	a set of objects identified by a *class word* together with a discriminating predicate. See **Classes**
combinator	a word taking a quotation or another word as input; a higher-order function. See **Combinators**
definition specifier	an instance of **definition** which implements the **Definition protocol**
generalized boolean	an object interpreted as a boolean; a value of **f** denotes false and anything else denotes true
generic word	a word whose behavior depends can be specialized on the class of one of its inputs. See **Generic words and methods**
method	a specialized behavior of a generic word on a class. See **Generic words and methods**
object	any datum which can be identified
ordering specifier	see **Ordering specifiers**
pathname string	an OS-specific pathname which identifies a file
quotation	an anonymous function; an instance of the **quotation** class. More generally, instances of the **callable** class can be used in many places documented to expect quotations
sequence	a sequence; see **Sequence protocol**
slot	a component of an object which can store a value
stack effect	a pictorial representation of a word's inputs and outputs, for example `+ ( x y -- z )`. See **Stack effect declarations**
true value	any object not equal to **f**
vocabulary **or** vocab	a named set of words. See **Vocabularies**
vocabulary specifier	a **vocab**, **vocab-link** or a string naming a vocabulary
word	the basic unit of code, analogous to a function or procedure in other programming languages. See **Words**

Documentation conventions

Factor documentation consists of two distinct bodies of text. There is a hierarchy of articles, much like this one, and there is word documentation. Help articles reference word documentation, and vice versa, but not every documented word is referenced from some help article.

The browser, completion popups and other tools use a common set of **Definition icons**.

Every article has links to parent articles at the top. Explore these if the article you are reading is too specific.

Some generic words have **Description** headings, and others have **Contract** headings. A distinction is made between words which are not intended to be extended with user-defined methods, and those that are.

Vocabulary naming conventions

A vocabulary name ending in `.private` contains words which are either implementation details, unsafe, or both. For example, the `sequences.private` vocabulary contains words which access sequence elements without bounds checking (**Unsafe sequence operations**). You should avoid using private words from the Factor library unless absolutely necessary. Similarly, your own code can place words in private vocabularies using **<PRIVATE** if you do not want other people using them without good reason.

Word naming conventions

These conventions are not hard and fast, but are usually a good first step in understanding a word's behavior:

General form	Description	Examples
`foo ?`	outputs a boolean	**empty?**
`foo !`	a variant of `foo` which mutates one of its arguments	**append!**
`?foo`	conditionally performs `foo`	**?nth**
`<foo >`	creates a new `foo`	**<array>**
`>foo`	converts the top of the stack into a `foo`	**>array**
`foo > bar`	converts a `foo` into a `bar`	**number>string**
`new-foo`	creates a new `foo`, taking some kind of parameter from the stack which determines the type of the object to be created	**new-sequence**, **new-lexer**, **new**
`foo *`	alternative form of `foo`, or a generic word called by `foo`	**at***, **pprint***
`(foo )`	implementation detail word used by `foo`	**(clone)**
`set-foo`	sets `foo` to a new value	**set-length**
`foo >>`	gets the `foo` slot of the tuple at the top of the stack; see **Slot accessors**	**name>>**
`>>foo`	sets the `foo` slot of the tuple at the top of the stack; see **Slot accessors**	**>>name**
`with-foo`	performs some kind of initialization and cleanup related to `foo`, usually in a new dynamic scope	**with-scope**, **with-input-stream**

,
with-output-stream

$foo help markup

$heading, **$emphasis**

Stack effect conventions

Stack effect conventions are documented in **Stack effect declarations**.

Syntax

Factor has two main forms of syntax: *definition* syntax and *literal* syntax. Code is data, so the syntax for code is a special case of object literal syntax. This section documents literal syntax. Definition syntax is covered in **Words**. Extending the parser is the main topic of **The parser**.

Parser algorithm

At the most abstract level, Factor syntax consists of whitespace-separated tokens. The parser tokenizes the input on whitespace boundaries. The parser is case-sensitive and whitespace between tokens is significant, so the following three expressions tokenize differently:

```
2X+
2 X +
2 x +
```

As the parser reads tokens it makes a distinction between numbers, ordinary words, and parsing words. Tokens are appended to the parse tree, the top level of which is a quotation returned by the original parser invocation. Nested levels of the parse tree are created by parsing words.

The parser iterates through the input text, checking each character in turn. Here is the parser algorithm in more detail -- some of the concepts therein will be defined shortly:

If the current character is a double-quote ("), the **"** parsing word is executed, causing a string to be read.

Otherwise, the next token is taken from the input. The parser searches for a word named by the token in the currently used set of vocabularies. If the word is found, one of the following two actions is taken:

If the word is an ordinary word, it is appended to the parse tree.

If the word is a parsing word, it is executed.

Otherwise if the token does not represent a known word, the parser attempts to parse it as a number. If the token is a number, the number object is added to the parse tree. Otherwise, an error is raised and parsing halts.

Parsing words play a key role in parsing; while ordinary words and numbers are simply added to the parse tree, parsing words execute in the context of the parser, and can do their own parsing and create nested data structures in the parse tree. Parsing words are also able to define new words.

While parsing words supporting arbitrary syntax can be defined, the default set is found in the **syntax** vocabulary and provides the basis for all further syntactic interaction with Factor.

Parse-time word lookup

When the parser reads a word name, it resolves the word at parse-time, looking up the **word** instance in the right vocabulary and adding it to the parse tree.

Initially, only words from the **syntax** vocabulary are available in source files. Since most files will use words in other vocabularies, they will need to make those words available using a set of parsing words.

Syntax to control word lookup
Private words
Resolution of ambiguous word names
Word lookup errors

See also

Words

Top level forms

Any code outside of a definition is known as a *top-level form*; top-level forms are run after the entire source file has been parsed, regardless of their position in the file.

Top-level forms cannot access the parse-time manifest (**Reflection support for vocabulary search path**), nor do they run inside **with-compilation-unit**; as a result, meta-programming might require extra work in a top-level form compared with a parsing word.

Also, top-level forms run in a new dynamic scope, so using **set** to store values is almost always wrong, since the values will be lost after the top-level form completes. To save values computed by a top-level form, either use **set-global** or define a new word with the value.

Comments

!

#!

Literals

Many different types of objects can be constructed at parse time via literal syntax. Numbers are a special case since support for reading them is built-in to the parser. All other literals are constructed via parsing words.

If a quotation contains a literal object, the same literal object instance is used each time the quotation executes; that is, literals are "live".

Using mutable object literals in word definitions requires care, since if those objects are mutated, the actual word definition will be changed, which is in most cases not what you would expect. Literals should be **clone**d before being passed to a word which may potentially mutate them.

Number syntax
Word syntax
Quotation syntax
Array syntax
Character and string syntax
Byte array syntax
Vector syntax

String buffer syntax
Hashtable syntax
Hash set syntax
Tuple syntax
Pathname syntax
Stack effect syntax

Parse time evaluation

Code can be evaluated at parse time. This is a rarely-used feature; one use-case is **Loading native libraries**, where you want to execute some code before the words in a source file are compiled.

<<

>> (-- *)

Stack machine model

Quotations are evaluated sequentially from beginning to end. When the end is reached, the quotation returns to its caller. As each object in the quotation is evaluated in turn, an action is taken based on its type:

a **word** - the word's definition quotation is called. See **Words**

a **wrapper** - the wrapped object is pushed on the data stack.
Wrappers are used to push word objects directly on the stack when they would otherwise execute. See the **** parsing word.

All other types of objects are pushed on the data stack.

Tail-call optimization

See also

Optimizing compiler

Tail-call optimization

If the last action performed is the execution of a word, the current quotation is not saved on the call stack; this is known as *tail-call optimization* and the Factor implementation guarantees that it will be performed.

Tail-call optimization allows iterative algorithms to be implemented in an efficient manner using recursion, without the need for any kind of primitive looping construct in the language. However, in practice, most iteration is performed via combinators such as **while**, **each**, **map**, **assoc-each**, and so on. The definitions of these combinators do bottom-out in recursive words, however.

Stack effect declarations

Word definition words such as **:** and **GENERIC:** have a *stack effect declaration* as part of their syntax. A stack effect declaration takes the following form:

```
( input1 input2 ... -- output1 ... )
```

Stack elements in a stack effect are ordered so that the top of the stack is on the right side. Here is an example:

```
IN: math MATH: + ( x y -- z )
```

Parameters which are quotations can be declared by suffixing the parameter name with `:` and then writing a nested stack effect declaration. If the number of inputs or outputs depends on the stack effects of quotation parameters, row variables can be declared:

Stack effect row variables

Some examples of row-polymorphic combinators:

```
IN: kernel : while ( ..a pred: ( ..a -- ..b ? ) body: ( ..b -- ..a
) -- ..b )
```

```
IN: kernel : if* ( ..a ? true: ( ..a ? -- ..b ) false: ( ..a -- ..b
) -- ..b )
```

```
IN: sequences : each ( ... seq quot: ( ... x -- ... ) -- ... )
```

For words that are not **inline**, only the number of inputs and outputs carries semantic meaning, and effect variables are ignored. However, nested quotation declarations are enforced for inline words. Nested quotation declarations are optional for non-recursive inline combinators and only provide better error messages. However, quotation inputs to **recursive** combinators must have an effect declared. See **Recursive combinator stack effects**.

In concatenative code, input and output names are for documentation purposes only and certain conventions have been established to make them more descriptive. For code written with **Lexical variables**, stack values are bound to local variables named by the stack effect's input parameters.

Inputs and outputs are typically named after some pun on their data type, or a description of the value's purpose if the type is very general. The following are some examples of value names:

`?`	a boolean
`<=>`	an ordering specifier; see **Ordering specifiers**
`elt`	an object which is an element of a sequence
`m`, `n`	an integer
`obj`	an object
`quot`	a quotation
`seq`	a sequence
`assoc`	an associative mapping
`str`	a string
`x`, `y`, `z`	a number
`loc`	a screen location specified as a two-element array holding x and y co-ordinates

`dim` a screen dimension specified as a two-element array holding width and height values

`*` when this symbol appears by itself in the list of outputs, it means the word unconditionally throws an error

For reflection and metaprogramming, you can use **Stack effect syntax** to include literal stack effects in your code, or these constructor words to construct stack effect objects at runtime:

<effect> (in out -- effect)

<terminated-effect> (in out terminated? -- effect)

<variable-effect> (in-var in out-var out -- effect)

See also

Stack effect checking

Stack effect checking

The **Optimizing compiler** checks the **Stack effect declarations** of words before they can be run. This ensures that words take exactly the number of inputs and outputs that the programmer declares in source.

Words that do not pass the stack checker are rejected and cannot be run, and so essentially this defines a very simple and permissive type system that nevertheless catches some invalid programs and enables compiler optimizations.

If a word's stack effect cannot be inferred, a compile error is reported. See **Compiler errors**.

The following articles describe how different control structures are handled by the stack checker.

Straight-line stack effects
Combinator stack effects
Recursive combinator stack effects
Branch stack effects

Stack checking catches several classes of errors.

Stack checker errors

Sometimes code with a dynamic stack effect has to be run.

Stack effect checking escape hatches

See also

Stack effect declarations, **Stack effect tools**, **Batch error reporting**

Straight-line stack effects

The simplest case is when a piece of code does not have any branches or recursion, and just pushes literals and calls words.

Pushing a literal has stack effect `( -- x )`. The stack effect of a most words is always known statically from the declaration. Stack effects of **inline** words and **Macros**, may depend on literals pushed on the stack prior to the call, and this case is discussed in **Combinator stack effects**.

The stack effect of each element in a code snippet is composed. The result is then the stack effect of the snippet.

An example:

```
[ 1 2 3 ] infer.
( -- x x x )
```

Another example:

```
[ 2 + ] infer.
( x -- x )
```

Combinator stack effects

If a word calls a combinator, one of the following two conditions must hold for the stack checker to succeed:

The combinator must be called with a quotation that is either literal or built from literal quotations, **curry**, and **compose**. (Note that quotations

that use **fry** or **locals** use **curry** and **compose** from the perspective of the stack checker.)

> If the word is declared **inline**, the combinator may additionally be called on one of the word's input parameters or with quotations built from the word's input parameters, literal quotations, **curry**, and **compose**. When inline, a word is itself considered to be a combinator, and its callers must in turn satisfy these conditions.

If neither condition holds, the stack checker throws a **unknown-macro-input** or **bad-macro-input** error. To make the code compile, a runtime checking combinator such as **call(** must be used instead. See **Stack effect checking escape hatches** for details. An inline combinator can be called with an unknown quotation by **curry**ing the quotation onto a literal quotation that uses **call(**.

Input stack effects

Inline combinators will verify the stack effect of their input quotations if they are declared in the combinator's stack effect. See **Stack effect row variables** for details.

Examples

Calling a combinator

The following usage of **map** passes the stack checker, because the quotation is the result of **curry**:

```
USING: math sequences ;
[ [ + ] curry map ] infer.
( x x -- x )
```

The equivalent code using **fry** and **locals** likewise passes the stack checker:

```
USING: fry math sequences ;
[ '[ _ + ] map ] infer.
( x x -- x )
```

```
USING: locals math sequences ;
[| a | [ a + ] map ] infer.
( x x -- x )
```

Defining an inline combinator

The following word calls a quotation twice; the word is declared **inline**, since it invokes **call** on the result of **compose** on an input parameter:

```
: twice ( value quot -- result ) dup compose call ; inline
```

The following code now passes the stack checker; it would fail were `twice` not declared **inline**:

```
USE: math.functions
[ [ sqrt ] twice ] infer.
( x -- x )
```

Defining a combinator for unknown quotations

In the next example, **call(** must be used because the quotation is the result of calling a runtime accessor, and the compiler cannot make any static assumptions about this quotation at all:

```
TUPLE: action name quot ;
: perform ( value action -- result ) quot>> call( value -- result )
;
```

Passing an unknown quotation to an inline combinator

Suppose we want to write:

```
: perform ( values action -- results ) quot>> map ;
```

However this fails to pass the stack checker since there is no guarantee the quotation has the right stack effect for **map**. It can be wrapped in a new quotation with a declaration:

```
: perform ( values action -- results )
    quot>> [ call( value -- result ) ] curry map ;
```

Explanation

This restriction exists because without further information, one cannot say what the stack effect of **call** is; it depends on the given quotation. If the stack checker encounters a **call** without further information, a **unknown-macro-input** or **bad-macro-input** error is raised.

On the other hand, the stack effect of applying **call** to a literal quotation or a **curry** of a literal quotation is easy to compute; it behaves as if the quotation was substituted at that point.

Limitations

The stack checker cannot guarantee that a literal quotation is still literal if it is passed on the data stack to an inlined recursive combinator such as **each** or **map**. For example, the following will not infer:

```
[ [ reverse ] swap [ reverse ] map swap call ] infer.
Cannot apply "call" to a run-time computed value
macro call
```

To make this work, use **dip** to pass the quotation instead:

```
[ [ reverse ] [ [ reverse ] map ] dip call ] infer.
( x -- x )
```

Recursive combinator stack effects

Most combinators do not call themselves recursively directly; instead, they are implemented in terms of existing combinators, for example **while**, **map**, and the **Compositional combinators**. In these cases, the rules outlined in **Combinator stack effects** apply.

Combinators which are recursive require additional care. In addition to being declared **inline**, they must be declared **recursive**. There are three restrictions that only apply to combinators with this declaration:

Input quotation declaration

Input parameters which are quotations must be annotated as much in the stack effect. For example, the following will not infer:

```
: bad ( quot -- ) [ call ] keep bad ; inline recursive
[ [ ] bad ] infer.
Cannot apply "call" to a run-time computed value
macro call
```

The following is correct:

```
: good ( quot: ( -- ) -- ) [ call ] keep good ; inline recursive
[ [ ] good ] infer.
( -- )
```

The effect of the nested quotation itself is only present for documentation purposes; the mere presence of a nested effect is sufficient to mark that value as a quotation parameter.

Data flow restrictions

The stack checker does not trace data flow in two instances.

An inline recursive word cannot pass a quotation on the data stack through the recursive call. For example, the following will not infer:

```
: bad ( ? quot: ( ? -- ) -- ) 2dup [ not ] dip bad call ; inline
```

```
recursive
[ [ drop ] bad ] infer.
Cannot apply "call" to a run-time computed value
macro call
```

However a small change can be made:

```
: good ( ? quot: ( ? -- ) -- ) [ good ] 2keep [ not ] dip call ;
inline recursive
[ [ drop ] good ] infer.
( x -- )
```

An inline recursive word must have a fixed stack effect in its base case. The following will not infer:

```
: foo ( quot ? -- ) [ f foo ] [ call ] if ; inline
[ [ 5 ] t foo ] infer.
The inline recursive word "foo" must be declared recursive
word foo
```

Branch stack effects

Conditionals such as **if** and combinators built on top have the same restrictions as **inline** combinators (see **Combinator stack effects**) with the additional requirement that all branches leave the stack at the same height. If this is not the case, the stack checker throws a **unbalanced-branches-error**.

If all branches leave the stack at the same height, then the stack effect of the conditional is just the maximum of the stack effect of each branch. For example,

```
[ [ + ] [ drop ] if ] infer.
( x x x -- x )
```

The call to **if** takes one value from the stack, a generalized boolean. The first branch `[ + ]` has stack effect `( x x -- x )` and the second has stack effect `( x -- )`. Since both branches decrease the height of the stack by one, we say that the stack effect of the two branches is `( x x -- x )`, and together with the boolean popped off the stack by **if**, this gives a total stack effect of `( x x x -- x )`.

Stack checker errors

Stack effect checking failure conditions are reported in one of two ways:

Stack effect tools report them when fed quotations interactively

The**Optimizing compiler** reports them while compiling words, via the **Batch error reporting** mechanism

Errors thrown when insufficient information is available to calculate the stack effect of a call to a combinator or macro (see **Combinator stack effects**):

do-not-compile (word -- *)

unknown-macro-input (macro -- *)

bad-macro-input (macro -- *)

Error thrown when a word's stack effect declaration does not match the composition of the stack effects of its factors:

effect-error (inferred declared -- *)

Error thrown when branches have incompatible stack effects (see **Branch stack effects**):

unbalanced-branches-error (word quots declareds actuals -- *)

Inference errors for inline recursive words (see **Recursive combinator stack effects**):

undeclared-recursion-error (word -- *)

diverging-recursion-error (word -- *)

unbalanced-recursion-error (word height -- *)

inconsistent-recursive-call-error (word -- *)

More obscure errors that are unlikely to arise in ordinary code:

recursive-quotation-error (quot -- *)

too-many->r (-- *)

too-many-r> (-- *)

missing-effect (word -- *)

Stack effect checking escape hatches

In a static checking regime, sometimes it is necessary to step outside the boundaries and run some code which cannot be statically checked; perhaps this code is constructed at run-time. There are two ways to get around the static stack checker.

If the stack effect of a word or quotation is known, but the word or quotation itself is not, **execute(** or **call(** can be used. See **Fundamental combinators** for details.

If the stack effect is not known, the code being called cannot manipulate the datastack directly. Instead, it must reflect the datastack into an array:

with-datastack (stack quot -- new-stack)

The surrounding code has a static stack effect since **with-datastack** has one. However, the array passed in as input may be transformed arbitrarily by calling this combinator.

Booleans

In Factor, any object that is not **f** has a true value, and **f** has a false value. The **t** object is the canonical true value.

f
t

A union class of the above:

boolean

There are some logical operations on booleans:

>boolean (obj -- ?)
not (obj -- ?)
and (obj1 obj2 -- ?)
or (obj1 obj2 -- ?)
xor (obj1 obj2 -- ?)

Boolean values are most frequently used for **Conditional combinators**.

The f object and f class

The**f** object is the unique instance of the **f** class; the two are distinct objects. The latter is also a parsing word which adds the **f** object to the parse tree at parse time. To refer to the class itself you must use **POSTPONE:** or **** to prevent the parsing word from executing.

Here is the **f** object:

```
f .
f
```

Here is the **f** class:

```
\ f .
POSTPONE: f
```

They are not equal:

```
f \ f = .
f
```

Here is an array containing the **f** object:

```
{ f } .
{ f }
```

Here is an array containing the **f** class:

```
{ POSTPONE: f } .
{ POSTPONE: f }
```

The**f** object is an instance of the **f** class:

```
USE: classes
f class-of .
POSTPONE: f
```

The**f** class is an instance of **word**:

```
USE: classes
\ f class-of .
word
```

On the other hand, **t** is just a word, and there is no class which it is a unique instance of.

```
t \ t eq? .
t
```

Many words which search collections confuse the case of no element being present with an element being found equal to **f**. If this distinction is important, there is usually an alternative word which can be used; for example, compare **at** with **at***.

Numbers

Number implementations:

Advanced features:

Arithmetic

Factor attempts to preserve natural mathematical semantics for numbers. Multiplying two large integers never results in overflow, and dividing two integers yields an exact ratio. Floating point numbers are also supported, along with complex numbers.

Math words are in the **math** vocabulary. Implementation details are in the **math.private** vocabulary.

See also

Integers, **Rational numbers**, **Floats**, **Complex numbers**

Constants

Standard mathematical constants:

e (-- e)

euler (-- gamma)

phi (-- phi)

pi (-- pi)

epsilon (-- epsilon)

single-epsilon (-- epsilon)

Mathematical functions

Powers and logarithms
Trigonometric and hyperbolic functions

Converting between numbers and strings

These words only convert between real numbers and strings. Complex numbers are constructed by the parser (**The parser**) and printed by the prettyprinter (**The prettyprinter**).

Integers can be converted to and from arbitrary bases. Floating point numbers can only be converted to and from base 10 and 16.

Converting numbers to strings:

number>string (n -- str)

>bin (n -- str)

>oct (n -- str)

>hex (n -- str)

>base (n radix -- str)

Converting strings to numbers:

string>number (str -- n/f)

bin> (str -- n/f)

oct> (str -- n/f)

hex> (str -- n/f)

base> (str radix -- n/f)

You can also input literal numbers in a different base (**Integer syntax**).

See also

Prettyprinting numbers

Integers

integer

Integers come in two varieties -- fixnums and bignums. Fixnums fit in a machine word and are faster to manipulate; if the result of a fixnum operation is too large to fit in a fixnum, the result is upgraded to a bignum. Here is an example where two fixnums are multiplied yielding a bignum:

```
USE: classes
67108864 class-of .
fixnum

USE: classes
128 class-of .
fixnum

134217728 128 * .
17179869184

USE: classes
1 128 shift class-of .
```

bignum

Integers can be entered using a different base; see **Number syntax**.

Integers can be tested for, and real numbers can be converted to integers:

fixnum? (object -- ?)

bignum? (object -- ?)

>fixnum (x -- n)

>integer (x -- n)

>bignum (x -- n)

See also

Prettyprinting numbers, **Modular arithmetic**, **Bitwise arithmetic**, **Integer functions**, **Integer syntax**

Rational numbers

ratio

When we add, subtract or multiply any two integers, the result is always an integer. However, dividing a numerator by a denominator that is not an integral divisor of the denominator yields a ratio:

```
1210 11 / .
110

100 330 / .
10/33

14 10 / .
1+2/5
```

Ratios are printed and can be input literally in the form above. Ratios are always reduced to lowest terms by factoring out the greatest common divisor of the numerator and denominator. A ratio with a denominator of 1 becomes an integer. Division with a denominator of 0 throws an error.

Ratios behave just like any other number -- all numerical operations work as you would expect.

```
1/2 1/3 + .
5/6

100 6 / 3 * .
50
```

Ratios can be taken apart:

numerator (a/b -- a)

denominator (a/b -- b)

>fraction (a/b -- a b)

See also

Ratio syntax

Floats

float

Rational numbers represent *exact* quantities. On the other hand, a floating point number is an *approximate* value. While rationals can grow to any required precision, floating point numbers have limited precision, and manipulating them is usually faster than manipulating ratios or bignums.

Introducing a floating point number in a computation forces the result to be expressed in floating point.

```
5/4 1/2 + .
1+3/4

5/4 0.5 + .
1.75
```

Floating point literal syntax is documented in **Float syntax**.

Integers and rationals can be converted to floats:

>float (x -- y)

Two real numbers can be divided yielding a float result:

/f (x y -- z)

Bitwise operations on floats
Floating point comparison operations

The**math.floats.env** vocabulary provides functionality for controlling floating point exceptions, rounding modes, and denormal behavior.

Complex numbers

complex

Complex numbers arise as solutions to quadratic equations whose graph does not intersect the *x* axis. Their literal syntax is covered in **Complex number syntax**.

Complex numbers can be taken apart:

real-part (z -- x)

imaginary-part (z -- y)

>rect (z -- x y)

Complex numbers can be constructed from real numbers:

rect> (x y -- z)

Embedding of real numbers in complex numbers

See also

Complex number syntax

Vector operations

Any Factor sequence can be used to represent a mathematical vector, however for best performance, the sequences defined by the **specialized-arrays** and **math.vectors.simd** vocabularies should be used.

Vector arithmetic
Vector component- and bit-wise logic
Vector shuffling, packing, and unpacking
Miscellaneous vector functions

Intervals

Interval arithmetic is performed on ranges of real numbers, rather than exact values. It is used by the Factor compiler to convert arbitrary-precision arithmetic to machine arithmetic, by inferring bounds for integer calculations.

Properties of interval arithmetic

The class of intervals:

interval

interval? (object -- ?)

Interval operations:

Creating intervals
Interval arithmetic
Set-theoretic operations on intervals
Comparing intervals

Collections

Sequences

Fixed-length sequences:

Resizable sequences:

Associative mappings

Implementations:

Double-ended queues

Implementations:

Other collections

There are also many other vocabularies tagged **collections** in the library.

Sequence operations

A *sequence* is a finite, linearly-ordered collection of elements. Words for working with sequences are in the **sequences** vocabulary.

Sequences implement a protocol:

Sequence protocol
The f object as a sequence

Sequence utility words can operate on any object whose class implements the sequence protocol. Most implementations are backed by storage. Some implementations obtain their elements from an underlying sequence, or compute them on the fly. These are known as **Virtual sequences**.

Accessing sequence elements
Sequence combinators
Adding and removing sequence elements
Appending sequences
Subsequences and slices
Reshaping sequences
Testing sequences
Searching sequences
Comparing sequences
Splitting sequences
Groups and clumps
Destructive sequence operations
Treating sequences as stacks
Sorting sequences
Binary search
Sets
Trimming sequences
Cartesian product operations
Deep sequence combinators

Using sequences for looping:

Counted loops
Numeric ranges

Using sequences for control flow:

Control flow with sequences

For inner loops:

Unsafe sequence operations

Implementing sequence combinators:

Implementing sequence combinators

Virtual sequences

A virtual sequence is an implementation of the **Sequence protocol** which does not store its own elements, and instead computes them, either from scratch or by retrieving them from another sequence.

Implementations include the following:

reversed

slice

Virtual sequences can be implemented with the **Virtual sequence protocol**, by translating an index in the virtual sequence into an index in another sequence.

See also

Counted loops

Making sequences with variables

The **make** vocabulary implements a facility for constructing **sequence**s and **assoc**s by holding a collector object in a variable. Storing the collector object in a variable rather than the stack may allow code to be written with less stack manipulation.

Object construction is wrapped in a combinator:

make (quot exemplar -- seq)

Inside the quotation passed to **make**, several words accumulate values:

, (elt --)

% (seq --)

(n --)

When making an **assoc**, you can use these words to add key/value pairs:

,, (value key --)

%% (assoc --)

The collector object can be accessed directly from inside a **make**:

building

```
USING: make math.parser ;
[ "Language #" % CHAR: \s , 5 # ] "" make print
Language # 5
```

Make philosophy

Arrays

The **arrays** vocabulary implements fixed-size mutable sequences which support the **Sequence protocol**.

The **arrays** vocabulary only includes words for creating new arrays. To access and modify array elements, use **Sequence operations** in the **sequences** vocabulary.

Array literal syntax is documented in **Array syntax**. Resizable arrays also exist and are known as **Vectors**.

Arrays form a class of objects:

array

array? (object -- ?)

Creating new arrays:

>array (seq -- array)

<array> (n elt -- array)

Creating an array from several elements on the stack:

1array (x -- array)

2array (x y -- array)

3array (x y z -- array)

4array (w x y z -- array)

Resizing arrays:

resize-array (n array -- new-array)

The class of two-element arrays:

pair

Arrays can be accessed without bounds checks in a pointer unsafe way.

Unsafe array operations

Quotations

A quotation is an anonymous function (a value denoting a snippet of code) which can be used as a value and called using the **Fundamental combinators**.

Quotation literals appearing in source code are delimited by square brackets, for example `[ 2 + ]`; see **Quotation syntax** for details on their syntax.

Quotations form a class of objects:

quotation

quotation? (object -- ?)

A more general class is provided for methods to dispatch on that includes quotations, **curry**, and **compose** objects:

callable

Quotations evaluate sequentially from beginning to end. Literals are pushed on the stack and words are executed. Details can be found in **Stack machine model**. Words can be placed in wrappers to suppress execution:

Wrappers

Quotations implement the **Sequence protocol**, and existing sequences can be converted into quotations:

>quotation (seq -- quot)

1quotation (obj -- quot)

Although quotations can be treated as sequences, the compiler will be unable to reason about quotations manipulated as sequences at runtime. **Compositional combinators** are provided for runtime partial application and composition of quotations.

Strings

The **strings** vocabulary implements a data type for storing text. Strings are represented as fixed-size mutable sequences of Unicode code points. Code points are represented as integers in the range [0,2,097,152].

Strings implement the **Sequence protocol**, and basic string manipulation can be performed with **Sequence operations** from the **sequences** vocabulary. More text processing functionality can be found in vocabularies carrying the **text** tag.

Strings form a class:

string

string? (object -- ?)

Creating new strings:

>string (seq -- str)

<string> (n ch -- string)

Creating a string from a single character:

1string (ch -- str)

Resizing strings:

resize-string (n str -- newstr)

See also

Character and string syntax, **String buffers**, **Unicode support**, **I/O encodings**

Byte arrays

Byte arrays are fixed-size mutable sequences (**Sequence protocol**) whose elements are integers in the range 0-255, inclusive. Each element only uses one byte of storage, hence the name. The literal syntax is covered in **Byte array syntax**.

Byte array words are in the **byte-arrays** vocabulary.

Byte arrays play a special role in the C library interface; they can be used to pass binary data back and forth between Factor and C. See **Passing pointers to C functions**.

Byte arrays form a class of objects.

byte-array

byte-array? (object -- ?)

There are several ways to construct byte arrays.

>byte-array (seq -- byte-array)

<byte-array> (n -- byte-array)

1byte-array (x -- byte-array)

2byte-array (x y -- byte-array)

3byte-array (x y z -- byte-array)

4byte-array (w x y z -- byte-array)

Resizing byte arrays:

resize-byte-array (n byte-array -- new-byte-array)

Specialized arrays

The **specialized-arrays** vocabulary implements fixed-length sequence types for storing machine values in a space-efficient manner without boxing.

A specialized array type needs to be generated for each element type. This is done with parsing words:

SPECIALIZED-ARRAY:

SPECIALIZED-ARRAYS:

This parsing word adds new words to the search path, documented in the next section.

Specialized array words
Passing specialized arrays to C functions
Vector arithmetic with specialized arrays
Specialized array examples

The **specialized-vectors** vocabulary provides a resizable version of this abstraction.

Vectors

The **vectors** vocabulary implements resizable mutable sequence which support the **Sequence protocol**.

The **vectors** vocabulary only includes words for creating new vectors. To access and modify vector elements, use **Sequence operations** in the **sequences** vocabulary.

Vector literal syntax is documented in **Vector syntax**.

Vectors are intended to be used with **Destructive sequence operations**. Code that does not modify sequences in-place can use fixed-size arrays without loss of generality; see **Arrays**.

Vectors form a class of objects:

vector

vector? (object -- ?)

Creating new vectors:

>vector (seq -- vector)

<vector> (n -- vector)

Creating a vector from a single element:

1vector (x -- vector)

If you don't care about initial capacity, an elegant way to create a new vector is to write:

```
V{ } clone
```

Byte vectors

The **byte-vectors** vocabulary implements resizable mutable sequence of unsigned bytes. Byte vectors implement the **Sequence protocol** and thus all **Sequence operations** can be used with them.

Byte vectors form a class:

byte-vector

byte-vector? (object -- ?)

Creating byte vectors:

>byte-vector (seq -- byte-vector)

<byte-vector> (n -- byte-vector)

Literal syntax:

BV{

If you don't care about initial capacity, a more elegant way to create a new byte vector is to write:

```
BV{ } clone
```

String buffers

The **sbufs** vocabulary implements resizable mutable sequence of characters. The literal syntax is covered in **String buffer syntax**.

String buffers implement the **Sequence protocol** and thus all **Sequence operations** can be used with them. String buffers can be used to construct new strings by accumulating substrings and characters, however usually they are only used indirectly, since the sequence construction words are more convenient to use in most cases (see **Making sequences with variables**).

String buffers form a class of objects:

sbuf

sbuf? (object -- ?)

Words for creating string buffers:

>sbuf (seq -- sbuf)

<sbuf> (n -- sbuf)

If you don't care about initial capacity, a more elegant way to create a new string buffer is to write:

```
SBUF" " clone
```

Resizable sequence implementation

Resizable sequences are implemented by having a wrapper object hold a reference to an underlying sequence, together with a fill pointer indicating how many elements of the underlying sequence are occupied. When the fill pointer exceeds the underlying sequence capacity, the underlying sequence grows.

There is a resizable sequence mixin:

growable

This mixin implements the sequence protocol by assuming the object has two specific slots:

`length` - the fill pointer (number of occupied elements in the underlying storage)

`underlying` - the underlying storage

The underlying sequence must implement a generic word:

resize (n seq -- newseq)

Vectors and **String buffers** are implemented using the resizable sequence facility.

Associative mapping operations

An *associative mapping*, abbreviated *assoc*, is a collection of key/value pairs which provides efficient lookup and storage indexed by key.

Words used for working with assocs are in the **assocs** vocabulary.

Associative mappings implement a protocol:

Associative mapping protocol

A large set of utility words work on any object whose class implements the associative mapping protocol.

Lookup and querying of assocs
Transposed assoc operations
Storing keys and values in assocs
Associative mapping combinators
Set-theoretic operations on assocs
Associative mapping conversions

Linked assocs

A *linked assoc* is an assoc which combines an underlying assoc with a dlist to form a structure which has the insertion and retrieval characteristics of the underlying assoc (typically a hashtable), but with the ability to get the entries in insertion order by calling **>alist**.

Linked assocs are implemented in the **linked-assocs** vocabulary.

linked-assoc

<linked-hash> (-- assoc)

<linked-assoc> (exemplar -- assoc)

Bidirectional assocs

A *bidirectional assoc* combines a pair of assocs to form a data structure where both normal assoc operations (eg, **at**), as well as **Transposed assoc operations** (eg, **value-at**) run in sub-linear time.

Bidirectional assocs implement the entire **Associative mapping protocol** with the exception of **delete-at**. Duplicate values are allowed, however value lookups with **value-at** only return the first key that a given value was stored with.

The class of biassocs:

biassoc

biassoc? (object -- ?)

Creating new biassocs:

<biassoc> (exemplar -- biassoc)

<bihash> (-- biassoc)

Converting existing assocs to biassocs:

>biassoc (assoc -- biassoc)

References

References provide a uniform way of accessing and changing values. Some examples of referenced values are variables, tuple slots, and keys or values of assocs. References can be read, written, and deleted. References are defined in the **refs** vocabulary, and new reference types can be made by implementing a protocol.

Reference protocol
Reference implementations
Reference utilities

References are used by the **UI inspector**.

Hashtables

A hashtable provides efficient (expected constant time) lookup and storage of key/value pairs. Keys are compared for equality, and a hashing function is used to reduce the number of comparisons made. The literal syntax is covered in **Hashtable syntax**.

Words for constructing hashtables are in the **hashtables** vocabulary. Hashtables implement the **Associative mapping protocol**, and all **Associative mapping operations** can be used on them; there are no hashtable-specific words to access and modify keys, because associative mapping operations are generic and work with all associative mappings.

Hashtables are a class of objects.

hashtable

hashtable? (object -- ?)

You can create a new hashtable with an initial capacity.

<hashtable> (n -- hash)

If you don't care about initial capacity, a more elegant way to create a new hashtable is to write:

```
H{ } clone
```

To convert an assoc to a hashtable:

>hashtable (assoc -- hashtable)

Further topics:

Hashtable keys
Hashtable utilities
Hashtable implementation details

Association lists

An *association list*, abbreviated *alist*, is an association represented as a sequence where all elements are key/value pairs. The **sequence** mixin is an instance of the **assoc** mixin, hence all sequences support the **Associative mapping protocol** in this way.

While not an association list, note that **f** also implements the associative mapping protocol in a trivial way; it is an immutable assoc with no entries.

An alist is slower to search than a hashtable for a large set of associations. The main advantage of an association list is that the elements are ordered; also sometimes it is more

convenient to construct an association list with sequence words than to construct a hashtable with association words. Much of the time, hashtables are more appropriate. See **Hashtables**.

There is no special syntax for literal alists since they are just sequences; in practice, literals look like so:

```
{
    { key1 value1 }
    { key2 value2 }
}
```

To make an assoc into an alist:

>alist (assoc -- newassoc)

Enumerations

An enumeration provides a view of a sequence as an assoc mapping integer indices to elements:

enum

<enum> (seq -- enum)

Inverting a permutation using enumerations:

```
IN: scratchpad
: invert ( perm -- perm' )
    <enum> sort-values keys ;
{ 2 0 4 1 3 } invert .
{ 1 3 0 4 2 }
```

Deques

The**deques** vocabulary implements the deque data structure which has constant-time insertion and removal of elements at both ends.

Deques must be instances of a mixin class:

deque

Deques must implement a protocol.

Querying the deque:

peek-front (deque -- obj)

peek-back (deque -- obj)

deque-empty? (deque -- ?)

deque-member? (value deque -- ?)

Adding and removing elements:

push-front* (obj deque -- node)

push-back* (obj deque -- node)

pop-front* (deque --)

pop-back* (deque --)

clear-deque (deque --)

Working with node objects output by **push-front*** and **push-back***:

delete-node (node deque --)

node-value (node -- value)

Utility operations built in terms of the above:

push-front (obj deque --)

push-all-front (seq deque --)

push-back (obj deque --)

push-all-back (seq deque --)

pop-front (deque -- obj)

pop-back (deque -- obj)

slurp-deque (deque quot --)

When using a deque as a queue, the convention is to queue elements with **push-front** and deque them with **pop-back**.

Double-linked lists

A double-linked list is the canonical implementation of a **deque**.

Double-linked lists form a class:

dlist

dlist? (object -- ?)

Constructing a double-linked list:

<dlist> (-- list)

Double-linked lists support all the operations of the deque protocol (**Deques**) as well as the following.

Iterating over elements:

dlist-each (... dlist quot: (... value -- ...) -- ...)

dlist-find (... dlist quot: (... value -- ... ?) -- ... obj/f ?)

dlist-filter (... dlist quot: (... value -- ... ?) -- ... dlist')

dlist-any? (... dlist quot: (... value -- ... ?) -- ... ?)

Deleting a node matching a predicate:

delete-node-if* (... dlist quot: (... value -- ... ?) -- ... obj/f ?)

delete-node-if (... dlist quot: (... value -- ... ?) -- ... obj/f)

Search deque implementation:

<hashed-dlist> (-- search-deque)

Search deques

A search deque is a data structure with constant-time insertion and removal of elements at both ends, and constant-time membership tests. Inserting an element more than once has no effect. Search deques implement all deque operations in terms of an underlying deque, and membership testing with **deque-member?** is implemented with an underlying assoc. Search deques are defined in the **search-deques** vocabulary.

Creating a search deque:

<search-deque> (assoc deque -- search-deque)

Sets

A set is an unordered list of elements. Words for working with sets are in the **sets** vocabulary.

All sets are instances of a mixin class:

set

set? (object -- ?)

Operations on sets
Set implementations

Lists

The**lists** vocabulary implements linked lists. There are simple strict linked lists, but a generic list protocol allows the implementation of lazy lists as well.

The list protocol
Constructing strict lists
Manipulating lists
Combinators for lists

Lazy lists

Disjoint sets

The**disjoint-sets** vocabulary implements the *disjoint set* data structure (also known as *union-find*, after the two main operations which it supports) that represents a set of elements partitioned into disjoint equivalence classes, or alternatively, an equivalence relation on a set.

The two main supported operations are equating two elements, which joins their equivalence classes, and checking if two elements belong to the same equivalence class. Both operations have the time complexity of the inverse Ackermann function, which for all intents and purposes is constant time.

The class of disjoint sets:

disjoint-set

Creating new disjoint sets:

<disjoint-set> (-- disjoint-set)

assoc>disjoint-set (assoc -- disjoint-set)

Queries:

equiv? (a b disjoint-set -- ?)

equiv-set-size (a disjoint-set -- n)

Adding elements:
add-atom (a disjoint-set --)

Equating elements:
equate (a b disjoint-set --)

Additionally, disjoint sets implement the **clone** generic word.

Interval maps

The**interval-maps** vocabulary implements a data structure, similar to assocs, where a set of closed intervals of keys are associated with values. As such, interval maps do not conform to the assoc protocol, because intervals of floats, for example, can be used, and it is impossible to get a list of keys in between.

The following operations are used to query interval maps:
interval-at* (key map -- value ?)

interval-at (key map -- value)

interval-key? (key map -- ?)

interval-values (map -- values)

Use the following to construct interval maps
<interval-map> (specification -- map)

coalesce (alist -- specification)

Heaps

A heap is an implementation of a *priority queue*, which is a structure that maintains a sorted set of elements. The key property is that insertion of an arbitrary element and removal of the first element (determined by order) is performed in O(log n) time.

Heap elements are key/value pairs and are compared using the **<=>** generic word on the first element of the pair.

There are two classes of heaps. Min-heaps sort their elements so that the minimum element is first:
min-heap

min-heap? (object -- ?)

<min-heap> (-- min-heap)

Max-heaps sort their elements so that the maximum element is first:
max-heap

max-heap? (object -- ?)

<max-heap> (-- max-heap)

Both obey a protocol.

Queries:

heap-empty? (heap -- ?)

heap-size (heap -- n)

heap-peek (heap -- value key)

Insertion:

heap-push (value key heap --)

heap-push* (value key heap -- entry)

heap-push-all (assoc heap --)

Removal:

heap-pop* (heap --)

heap-pop (heap -- value key)

heap-delete (entry heap --)

Processing heaps:

slurp-heap (heap quot: (elt --) --)

Boxes

A *box* is a container which can either be empty or hold a single value.

box

Creating an empty box:

<box> (-- box)

Storing a value and removing a value from a box:

>box (value box --)

box> (box -- value)

Safely removing a value:

?box (box -- value/f ?)

Testing if a box is full can be done by reading the `occupied` slot.

Directed graph utilities

Words for treating associative mappings as directed graphs can be found in the **graphs** vocabulary. A directed graph is represented as an assoc mapping each vertex to a set of edges entering that vertex, where the set is itself an assoc, with equal keys and values.

To create a new graph, just create an assoc, for example by calling **<hashtable>**. To add vertices and edges to a graph:

add-vertex (vertex edges graph --)

To remove vertices from the graph:

remove-vertex (vertex edges graph --)

Since graphs are represented as assocs, they can be cleared out by calling **clear-assoc**.

You can perform queries on the graph:

closure (obj quot -- assoc)

Directed graphs are used to maintain cross-referencing information for **Definitions**.

Locked I/O buffers

I/O buffers are first-in-first-out queues of bytes.

Buffers are backed by manually allocated storage that does not get moved by the garbage collector; they are also low-level and sacrifice error checking for efficiency.

Buffers are used to implement native I/O backends.

Buffer words are found in the **io.buffers** vocabulary.

buffer

<buffer> (n -- buffer)

Buffers must be manually deallocated by calling **dispose**.

Buffer operations:

buffer-reset (n buffer --)

buffer-length (buffer -- n)

buffer-empty? (buffer -- ?)

buffer-capacity (buffer -- n)

buffer@ (buffer -- alien)

Reading from the buffer:

buffer-peek (buffer -- byte)

buffer-pop (buffer -- byte)

buffer-read (n buffer -- byte-array)

Writing to the buffer:

byte>buffer (byte buffer --)

>buffer (byte-array buffer --)

n>buffer (n buffer --)

Words

Words are the Factor equivalent of functions or procedures in other languages. Words are essentially named **Quotations**.

There are two ways of creating word definitions:

using parsing words at parse time.

using defining words at run time.

The latter is a more dynamic feature that can be used to implement code generation and such, and in fact parse time defining words are implemented in terms of run time defining words.

Types of words:

Colon definitions
Symbols
Word aliasing
Constants
Primitives

Advanced topics:

Deferred words and mutual recursion
Compiler declarations
Word introspection

See also

Vocabularies, **Vocabulary loader**, **Definitions**, **Printing definitions**

Colon definitions

All words have associated definition **Quotations**. A word's definition quotation is called when the word is executed. A *colon definition* is a word where this quotation is supplied directly by the user. This is the simplest and most common type of word definition.

Defining words at parse time:

:

; (-- *)

Defining words at run time:

define (word def --)

define-declared (word def effect --)

define-inline (word def effect --)

Word definitions must declare their stack effect. See **Stack effect declarations**.

All other types of word definitions, such as **Symbols** and **Generic words and methods**, are just special cases of the above.

Symbols

A symbol pushes itself on the stack when executed. By convention, symbols are used as

variable names (**Dynamic variables**).

symbol

symbol? (object -- ?)

Defining symbols at parse time:

SYMBOL:

SYMBOLS:

Defining symbols at run time:

define-symbol (word --)

Symbols are just compound definitions in disguise. The following two lines are equivalent:

```
SYMBOL: foo
: foo ( -- value ) \ foo ;
```

Word aliasing

There is a syntax for defining new names for existing words. This useful for C library bindings, for example in the Win32 API, where words need to be renamed for symmetry.

Define a new word that aliases another word:

ALIAS:

Define an alias at run-time:

define-alias (new old --)

Constants

There is a syntax for defining words which push literals on the stack.

Define a new word that pushes a literal on the stack:

CONSTANT:

Define an constant at run-time:

define-constant (word value --)

Primitives

Primitives are words defined in the Factor VM. They provide the essential low-level services to the rest of the system.

primitive

primitive? (object -- ?)

Deferred words and mutual recursion

Words cannot be referenced before they are defined; that is, source files must order definitions in a strictly bottom-up fashion. This is done to simplify the implementation, facilitate better parse time checking and remove some odd corner cases; it also encourages better coding style.

Sometimes this restriction gets in the way, for example when defining mutually-recursive words; one way to get around this limitation is to make a forward definition.

DEFER:

The class of deferred word definitions:

deferred

deferred? (object -- ?)

Deferred words throw an error when called:

undefined (-- *)

Deferred words are just compound definitions in disguise. The following two lines are equivalent:

```
DEFER: foo
: foo ( -- * ) undefined ;
```

Compiler declarations

Compiler declarations are parsing words that set a word property in the most recently defined word. They appear after the final **;** of a word definition:

```
: cubed ( x -- y ) dup dup * * ; foldable
```

Compiler declarations assert that the word follows a certain contract, enabling certain optimizations that are not valid in general.

inline

foldable

flushable

recursive

It is entirely up to the programmer to ensure that the word satisfies the contract of a declaration. Furthermore, if a generic word is declared **foldable** or **flushable**, all methods must satisfy the contract. Unspecified behavior may result if a word does not follow the contract of one of its declarations.

See also

Stack effect declarations

Word introspection

Word introspection facilities and implementation details are found in the **words** vocabulary.

Word objects contain several slots:

`name`	a word name
`vocabulary`	a word vocabulary name
`def`	a definition quotation
`props`	an assoc of word properties, including documentation and other meta-data

Words are instances of a class.

word(-- word)

word? (object -- ?)

Words implement the definition protocol; see **Definitions**.

Looking up and creating words

Uninterned words

Word properties

Word implementation details

Shuffle words

Shuffle words rearrange items at the top of the data stack as indicated by their stack effects. They provide simple data flow control between words. More complex data flow control is available with the **Dataflow combinators** and with **Lexical variables**.

Removing stack elements:

drop (x --)

2drop (x y --)

3drop (x y z --)

nip (x y -- y)

2nip (x y z -- z)

Duplicating stack elements:

dup (x -- x x)

2dup (x y -- x y x y)

3dup (x y z -- x y z x y z)

over (x y -- x y x)

2over (x y z -- x y z x y)

pick (x y z -- x y z x)

Permuting stack elements:

swap (x y -- y x)

There are additional, more complex stack shuffling words whose use is not recommended.

Complex shuffle words

Complex shuffle words

These shuffle words tend to make code difficult to read and to reason about. Code that uses them should almost always be rewritten using **Lexical variables** or **Dataflow combinators**.

Duplicating stack elements deep in the stack:

dupd (x y -- x x y)

Permuting stack elements deep in the stack:

swapd (x y z -- y x z)

rot (x y z -- y z x)

-rot (x y z -- z x y)

Combinators

A central concept in Factor is that of a *combinator*, which is a word taking code as input.

More combinators are defined for working on data structures, such as **Sequence combinators** and **Associative mapping combinators**.

See also

Quotations

Fundamental combinators

The most basic combinators are those that take either a quotation or word, and invoke it immediately. There are two sets of these fundamental combinators. They differ in whether the compiler is expected to determine the stack effect of the expression at compile time or the stack effect is declared and verified at run time.

Compile-time checked combinators

With these combinators, the compiler attempts to determine the stack effect of the expression at compile time, rejecting the program if the effect cannot be determined. See **Combinator stack effects**.

call (callable --)

execute (word --)

Run-time checked combinators

With these combinators, the stack effect of the expression is checked at run time.

call(

execute(

Note that the opening parenthesis is actually part of the word name for `call(` and `execute(`; they are parsing words, and they read a stack effect until the corresponding closing parenthesis. The underlying words are a bit more verbose, but they can be given non-constant stack effects:

call-effect (quot effect --)

execute-effect (word effect --)

Unchecked combinators

Unsafe combinators

See also

Stack effect declarations, **Stack effect checking**

Dataflow combinators

Dataflow combinators express common dataflow patterns such as performing a operation while preserving its inputs, applying multiple operations to a single value, applying a set of operations to a set of values, or applying a single operation to multiple values.

Preserving combinators
Cleave combinators
Spread combinators
Apply combinators

More intricate dataflow can be constructed by composing **Curried dataflow combinators**.

Conditional combinators

The basic conditionals:

if (..a ? true: (..a -- ..b) false: (..a -- ..b) -- ..b)

when (..a ? true: (..a -- ..a) -- ..a)

unless (..a ? false: (..a -- ..a) -- ..a)

Forms abstracting a common stack shuffle pattern:

if* (..a ? true: (..a ? -- ..b) false: (..a -- ..b) -- ..b)

when* (..a ? true: (..a ? -- ..a) -- ..a)

unless* (..a ? false: (..a -- ..a x) -- ..a x)

Another form abstracting a common stack shuffle pattern:

?if (..a default cond true: (..a cond -- ..b) false: (..a default -- ..b) -- ..b)

Sometimes instead of branching, you just need to pick one of two values:

? (? true false -- true/false)

Two combinators which abstract out nested chains of **if**:

cond (assoc --)

case (obj assoc --)

Expressing conditionals with boolean logic

See also

Booleans, **Bitwise arithmetic**, **both?**, **either?**

Looping combinators

In most cases, loops should be written using high-level combinators (such as **Sequence combinators**) or tail recursion. However, sometimes, the best way to express intent is with a loop.

while (..a pred: (..a -- ..b ?) body: (..b -- ..a) -- ..b)

until (..a pred: (..a -- ..b ?) body: (..b -- ..a) -- ..b)

To execute one iteration of a loop, use the following word:

do (pred body -- pred body)

This word is intended as a modifier. The normal **while** loop never executes the body if the predicate returns false on the first iteration. To ensure the body executes at least once, use **do**:

```
[ P ] [ Q ] do while
```

A simpler looping combinator which executes a single quotation until it returns **f**:

loop (... pred: (... -- ... ?) -- ...)

Compositional combinators

Certain combinators transform quotations to produce a new quotation.

Examples of compositional combinator usage

Fundamental operations:

curry (obj quot -- curry)

compose (quot1 quot2 -- compose)

Derived operations:

2curry (obj1 obj2 quot -- curry)

3curry (obj1 obj2 obj3 quot -- curry)

with (param obj quot -- obj curry)

prepose (quot1 quot2 -- compose)

These operations run in constant time, and in many cases are optimized out altogether by the **Optimizing compiler**. **Fried quotations** are an abstraction built on top of these operations, and code that uses this abstraction is often clearer than direct calls to the below words.

Curried dataflow combinators can be used to build more complex dataflow by combining cleave, spread and apply patterns in various ways.

Curried dataflow combinators

Quotations also implement the sequence protocol, and can be manipulated with sequence words; see **Quotations**. However, such runtime quotation manipulation will not be optimized by the optimizing compiler.

Short-circuit combinators

The**combinators.short-circuit** vocabulary stops a computation early once a condition is met.

AND combinators:

0&& (quots -- ?)

1&& (obj quots -- ?)

2&& (obj1 obj2 quots -- ?)

3&& (obj1 obj2 obj3 quots -- ?)

OR combinators:

0|| (quots -- ?)

1|| (obj quots -- ?)

2|| (obj1 obj2 quots -- ?)

3|| (obj1 obj2 obj3 quots -- ?)

Generalized combinators:

n&& (quots n -- quot)

n|| (quots n -- quot)

Smart combinators

A *smart combinator* is a macro which reflects on the stack effect of an input quotation. The **combinators.smart** vocabulary implements a few simple smart combinators which look at the static stack effects of input quotations and generate code which produces or consumes the relevant number of stack values.

Take all input values from a sequence:

input<sequence (seq quot --)

input<sequence-unsafe (seq quot --)

Store all output values to a sequence:

output>sequence (quot exemplar -- seq)

output>array (quot -- array)

Reducing the set of output values:

reduce-outputs (quot operation --)

map-reduce-outputs (quot mapper reducer -- quot)

Applying a quotation to groups of elements on the stack:

smart-apply (quot n --)

Summing output values:

sum-outputs (quot -- n)

Concatenating output values:

append-outputs (quot -- seq)

append-outputs-as (quot exemplar -- seq)

Drop the outputs after calling a quotation:

drop-outputs (quot --)

Cause a quotation to act as a no-op and drop the inputs:

nullary (quot --)

Preserve the inputs below or above the outputs of the quotation:

preserving (quot --)

keep-inputs (quot --)

Versions of if that infer how many inputs to keep from the predicate quotation:

smart-if (pred true false --)

smart-when (pred true --)

smart-unless (pred false --)

Versions of if* that infer how many inputs to keep from the predicate quotation:

smart-if* (pred true false --)

smart-when* (pred true --)

smart-unless* (pred false --)

New smart combinators can be created by defining **Macros** which call **infer**.

Quotation construction utilities

Some words for creating quotations which can be useful for implementing method combinations and compiler transforms:

cond>quot (assoc -- quot)

case>quot (default assoc -- quot)

alist>quot (default assoc -- quot)

Generalized shuffle words and combinators

The**generalizations** vocabulary defines a number of stack shuffling words and combinators for use in macros where the arity of the input quotations depends on an input parameter.

Generalized shuffle words
Generalized combinators
Additional generalizations

Also see the **sequences.generalizations** vocabulary for generalized sequence operations.

Co-operative threads

Factor supports co-operative threads. A thread will yield while waiting for input/output operations to complete, or when a yield has been explicitly requested.

Words for working with threads are in the **threads** vocabulary.

Starting and stopping threads

Spawning new threads:

spawn (quot name -- thread)

spawn-server (quot name -- thread)

Creating and spawning a thread can be factored out into two separate steps:

<thread> (quot name -- thread)

(spawn) (thread --)

Threads stop either when the quotation given to **spawn** returns, or when the following word is called:

stop (-- *)

If the image is saved and started again, all runnable threads are stopped. Vocabularies wishing to have a background thread always running should use **add-startup-hook**.

Yielding and suspending threads

Yielding to other threads:

yield (--)

Sleeping for a period of time:

sleep (dt --)

Interrupting sleep:

interrupt (thread --)

Threads can be suspended and woken up at some point in the future when a condition is satisfied:

suspend (state -- obj)

resume (thread --)

resume-with (obj thread --)

Thread-local state and variables

Threads form a class of objects:

thread

The current thread:

self (-- thread)

Thread-local variables:

tnamespace (-- assoc)

tget (key -- value)

tset (value key --)

tchange (..a key quot: (..a value -- ..b newvalue) -- ..b)

Each thread has its own independent set of thread-local variables and newly-spawned threads begin with an empty set.

Global hashtable of all threads, keyed by `id`:

threads (-- assoc)

Threads have an identity independent of continuations. If a continuation is reified in one thread and then reflected in another thread, the code running in that continuation will observe a change in the value output by **self**.

Thread implementation

Thread implementation:

run-queue (-- dlist)

sleep-queue (-- heap)

Lexical variables

The **locals** vocabulary provides lexically scoped local variables. Full closure semantics, both downward and upward, are supported. Mutable variable bindings are also provided, supporting assignment to bindings in the current scope or in outer scopes.

Examples of lexical variables

Word definitions where the inputs are bound to lexical variables:

::

M::

MEMO::

MACRO::

Lexical scoping and binding forms:

[let

:>

Quotation literals where the inputs are bound to lexical variables:

[|

Additional topics:

Lexical variables in literals
Mutable lexical variables
Lexical variables and fry
Limitations of lexical variables

Lexical variables complement **Dynamic variables**.

Examples of lexical variables

Definitions with lexical variables

The following example demonstrates lexical variable bindings in word definitions. The `quadratic-roots` word is defined with **::**, so it takes its inputs from the top three elements of the datastack and binds them to the variables `a`, `b`, and `c`. In the body, the `disc` variable is bound using **:>** and then used in the following line of code.

```
USING: locals math math.functions kernel ;
IN: scratchpad
:: quadratic-roots ( a b c -- x y )
    b sq 4 a c * * - sqrt :> disc
    b neg disc [ + ] [ - ] 2bi [ 2 a * / ] bi@ ;
1.0 1.0 -6.0 quadratic-roots [ . ] bi@
2.0
-3.0
```

If you wanted to perform the quadratic formula interactively from the listener, you could use **[let** to provide a scope for the variables:

```
USING: locals math math.functions kernel ;
IN: scratchpad
```

```
[let 1.0 :> a 1.0 :> b -6.0 :> c
    b sq 4 a c * * - sqrt :> disc
    b neg disc [ + ] [ - ] 2bi [ 2 a * / ] bi@
] [ . ] bi@
2.0
-3.0
```

Quotations with lexical variables, and closures

These next two examples demonstrate lexical variable bindings in quotations defined with **[|**. In this example, the values `5` and `3` are put on the datastack. When the quotation is called, it takes those values as inputs and binds them respectively to `m` and `n` before executing the quotation:

```
USING: kernel locals math prettyprint ;
IN: scratchpad
5 3 [| m n | m n - ] call .
2
```

In this example, the `adder` word creates a quotation that closes over its argument `n`. When called, the result quotation of `5 adder` pulls `3` off the datastack and binds it to `m`, which is added to the value `5` bound to `n` in the outer scope of `adder`:

```
USING: kernel locals math prettyprint ;
IN: scratchpad
:: adder ( n -- quot ) [| m | m n + ] ;
3 5 adder call .
8
```

Mutable bindings

This next example demonstrates closures and mutable variable bindings. The `<counter>` word outputs a tuple containing a pair of quotations that respectively increment and decrement an internal counter in the mutable `value` variable and then return the new value. The quotations close over the counter, so each invocation of the word gives new quotations with a new internal counter.

```
USING: locals kernel math ;
IN: scratchpad

TUPLE: counter adder subtractor ;

:: <counter> ( -- counter )
    0 :> value!
    counter new
    [ value 1 + dup value! ] >>adder
    [ value 1 - dup value! ] >>subtractor ;
<counter>
[ adder>>      call . ]
[ adder>>      call . ]
[ subtractor>> call . ] tri
1
2
1
```

The same variable name can be bound multiple times in the same scope. This is different from reassigning the value of a mutable variable. The most recent binding for a variable name will mask previous bindings for that name. However, the old binding referring to the previous value can still persist in closures. The following contrived example demonstrates this:

```
USING: kernel locals prettyprint ;
```

```
IN: scratchpad
:: rebinding-example ( -- quot1 quot2 )
    5 :> a [ a ]
    6 :> a [ a ] ;
:: mutable-example ( -- quot1 quot2 )
    5 :> a! [ a ]
    6 a! [ a ] ;
rebinding-example [ call . ] bi@
mutable-example [ call . ] bi@
5
6
6
6
```

In `rebinding-example`, the binding of `a` to `5` is closed over in the first quotation, and the binding of `a` to `6` is closed over in the second, so calling both quotations results in `5` and `6` respectively. By contrast, in `mutable-example`, both quotations close over a single binding of `a`. Even though `a` is assigned to `6` after the first quotation is made, calling either quotation will output the new value of `a`.

Lexical variables in literals

Some kinds of literals can include references to lexical variables as described in **Lexical variables in literals**. For example, the **3array** word could be implemented as follows:

```
USING: locals prettyprint ;
IN: scratchpad

:: my-3array ( x y z -- array ) { x y z } ;
1 "two" 3.0 my-3array .
{ 1 "two" 3.0 }
```

Lexical variables in literals

Certain data type literals are permitted to contain lexical variables. Any such literals are rewritten into code which constructs an instance of the type with the values of the variables spliced in. Conceptually, this is similar to the transformation applied to quotations containing free variables.

The data types which receive this special handling are the following:

- **Arrays**
- **Hashtables**
- **Vectors**
- **Tuples**
- **Wrappers**

Object identity

This feature changes the semantics of literal object identity. An ordinary word containing a literal pushes the same literal on the stack every time it is invoked:

```
IN: scratchpad
TUPLE: person first-name last-name ;
: ordinary-word-test ( -- tuple )
    T{ person { first-name "Alan" } { last-name "Kay" } } ;
ordinary-word-test ordinary-word-test eq? .
t
```

Inside a lexical scope, literals which do not contain lexical variables still behave in the same way:

```
USE: locals
IN: scratchpad
TUPLE: person first-name last-name ;
:: locals-word-test ( -- tuple )
    T{ person { first-name "Alan" } { last-name "Kay" } } ;
locals-word-test locals-word-test eq? .
t
```

However, literals with lexical variables in them actually construct a new object:

```
USING: locals splitting ;
IN: scratchpad
TUPLE: person first-name last-name ;
:: constructor-test ( -- tuple )
    "Jane Smith" " " split1 :> last :> first
    T{ person { first-name first } { last-name last } } ;
constructor-test constructor-test eq? .
f
```

One exception to the above rule is that array instances containing free lexical variables (that is, immutable lexical variables not referenced in a closure) do retain identity. This allows macros such as **cond** to expand at compile time even when their arguments reference variables.

Mutable lexical variables

When a lexical variable is bound using **:>**, **::**, or **[|**, the variable may be made mutable by suffixing its name with an exclamation point (`!`). A mutable variable's value is read by giving its name without the exclamation point as usual. To write to the variable, use its name with the `!` suffix.

Mutable bindings are implemented in a manner similar to that taken by the ML language. Each mutable binding is actually an immutable binding of a mutable cell. Reading the binding automatically unboxes the value from the cell, and writing to the binding stores into it.

Writing to mutable variables from outer lexical scopes is fully supported and has full closure semantics. See **Examples of lexical variables** for examples of mutable lexical variables in action.

Lexical variables and fry

Lexical variables integrate with **Fried quotations** so that mixing variables with fried quotations gives intuitive results.

The following two code snippets are equivalent:

```
'[ sq _ + ]
```

```
[ [ sq ] dip + ] curry
```

The semantics of **dip** and **curry** are such that the first example behaves as if the top of the stack as "inserted" in the "hole" in the quotation's second element.

Conceptually, **curry** is defined so that the following two code snippets are equivalent:

```
3 [ - ] curry
```

```
[ 3 - ]
```

When quotations take named parameters using **[|**, **curry** fills in the variable bindings from right to left. The following two snippets are equivalent:

```
3 [| a b | a b - ] curry
```

```
[| a | a 3 - ]
```

Because of this, the behavior of `fry` changes when applied to such a quotation to ensure that fry conceptually behaves the same as with normal quotations, placing the fried values "underneath" the variable bindings. Thus, the following snippets are no longer equivalent:

```
'[ [| a | _ a - ] ]
```

```
'[ [| a | a - ] curry ] call
```

Instead, the first line above expands into something like the following:

```
[ [ swap [| a | a - ] ] curry call ]
```

The precise behavior is as follows. When frying a **[|** quotation, a stack shuffle (**mnswap**) is prepended so that the `m` curried values, which start off at the top of the stack, are transposed with the quotation's `n` named input bindings.

Limitations of lexical variables

There are two main limitations of the current implementation, and both concern macros.

Macro expansions with free variables

The expansion of a macro cannot reference lexical variables bound in the outer scope. For example, the following macro is invalid:

```
MACRO:: twice ( quot -- ) [ quot call quot call ] ;
```

The following is fine, though:

```
MACRO:: twice ( quot -- ) quot quot '[ @ @ ] ;
```

Static stack effect inference and macros

A macro will only expand at compile-time if all of its inputs are literal. Likewise, the word containing the macro will only have a static stack effect and compile successfully if the macro's inputs are literal. When lexical variables are used in a macro's literal arguments, there is an additional restriction: The literals must immediately precede the macro call lexically.

For example, all of the following three code snippets are superficially equivalent, but only the first will compile:

```
:: good-cond-usage ( a -- ... )
    {
        { [ a 0 < ] [ ... ] }
        { [ a 0 > ] [ ... ] }
        { [ a 0 = ] [ ... ] }
    } cond ;
```

The next two snippets will not compile because the argument to **cond** does not immediately precede the call:

```
: my-cond ( alist -- ) cond ; inline

:: bad-cond-usage ( a -- ... )
    {
        { [ a 0 < ] [ ... ] }
        { [ a 0 > ] [ ... ] }
        { [ a 0 = ] [ ... ] }
    } my-cond ;

:: bad-cond-usage ( a -- ... )
    {
        { [ a 0 < ] [ ... ] }
        { [ a 0 > ] [ ... ] }
        { [ a 0 = ] [ ... ] }
    } swap swap cond ;
```

The reason is that lexical variable references are rewritten into stack code at parse time, whereas macro expansion is performed later during compile time. To circumvent this problem, the **macros.expander** vocabulary is used to rewrite simple macro usages prior to lexical variable transformation. However, **macros.expander** cannot deal with more complicated cases where the literal inputs to the macro do not immediately precede the macro call in the source.

Dynamic variables

The **namespaces** vocabulary implements dynamically-scoped variables.

A dynamic variable is an entry in an assoc of bindings, where the assoc is implicit rather than passed on the stack. These assocs are termed *namespaces*. Nesting of scopes is implemented with a search order on namespaces, defined by a *namestack*. Since namespaces are just assocs, any object can be used as a variable. By convention, variables are keyed by **Symbols**.

The **get** and **set** words read and write variable values. The **get** word searches the chain of nested namespaces, while **set** always sets variable values in the current namespace only. Namespaces are dynamically scoped; when a quotation is called from a nested scope, any words called by the quotation also execute in that scope.

get (variable -- value)

set (value variable --)

Various utility words provide common variable access patterns:

Changing variable values
Namespace combinators

Implementation details your code probably does not care about:

Namespace implementation details

Dynamic variables complement **Lexical variables**.

Changing variable values

on (variable --)

off (variable --)

inc (variable --)

dec (variable --)

change (variable quot --)

change-global (variable quot --)

toggle (variable --)

Namespace combinators

make-assoc (quot exemplar -- hash)

with-scope (quot --)

with-variable (value key quot --)

with-variables (ns quot --)

Namespace implementation details

The namestack holds namespaces.

namestack (-- namestack)

set-namestack (namestack --)

namespace (-- namespace)

A pair of words push and pop namespaces on the namestack.

>n (namespace --)

ndrop (--)

Global variables

namespace (-- namespace)

global (-- g)

get-global (variable -- value)

set-global (value variable --)

initialize (variable quot --)

with-global (quot --)

Fried quotations

The**fry** vocabulary implements *fried quotation*. Conceptually, fried quotations are quotations with "holes" (more formally, *fry specifiers*), and the holes are filled in when the fried quotation is pushed on the stack.

Fried quotations are started by a special parsing word:

’[

There are two types of fry specifiers; the first can hold a value, and the second "splices" a quotation, as if it were inserted without surrounding brackets:

_ (-- *)

@ (-- *)

The holes are filled in with the top of stack going in the rightmost hole, the second item on the stack going in the second hole from the right, and so on.

Examples of fried quotations
Fried quotation philosophy

Fry is implemented as a parsing word which reads a quotation and scans for occurrences of _ and @; these words are not actually executed, and doing so raises an error (this can happen if they’re accidentally used outside of a fry).

Fried quotations can also be constructed without using a parsing word; this is useful when meta-programming:

fry (quot -- quot’)

Fried quotations are an abstraction on top of the **Compositional combinators**; their use is encouraged over the combinators, because often the fry form is shorter and clearer than the combinator form.

Examples of fried quotations

The easiest way to understand fried quotations is to look at some examples.

If a quotation does not contain any fry specifiers, then **’[** behaves just like **[**:

```
{ 10 20 30 } ’[ . ] each
```

Occurrences of _ on the left map directly to **curry**. That is, the following three lines are equivalent:

```
{ 10 20 30 } 5 ’[ _ + ] map
{ 10 20 30 } 5 [ + ] curry map
{ 10 20 30 } [ 5 + ] map
```

Occurrences of _ in the middle of a quotation map to more complex quotation composition patterns. The following three lines are equivalent:

```
{ 10 20 30 } 5 ’[ 3 _ / ] map
{ 10 20 30 } 5 [ 3 ] swap [ / ] curry compose map
{ 10 20 30 } [ 3 5 / ] map
```

Occurrences of @ are simply syntax sugar for `_ call`. The following four lines are equivalent:

```
{ 10 20 30 } [ sq ] ’[ @ . ] each
```

```
{ 10 20 30 } [ sq ] [ call . ] curry each
{ 10 20 30 } [ sq ] [ . ] compose each
{ 10 20 30 } [ sq . ] each
```

The _ and @ specifiers may be freely mixed, and the result is considerably more concise and readable than the version using **curry** and **compose** directly:

```
{ 8 13 14 27 } [ even? ] 5 ’[ @ dup _ ? ] map
{ 8 13 14 27 } [ even? ] 5 [ dup ] swap [ ? ] curry compose compose
map
{ 8 13 14 27 } [ even? dup 5 ? ] map
```

The following is a no-op:

```
’[ @ ]
```

Here are some built-in combinators rewritten in terms of fried quotations:

```
literalize : literalize ’[ _ ] ;
curry     : curry ’[ _ @ ] ;
compose   : compose ’[ @ @ ] ;
```

Fried quotation philosophy

Fried quotations generalize quotation-building words such as **curry** and **compose**. They can clean up code with lots of currying and composition, particularly when quotations are nested:

```
’[ [ _ key? ] all? ] filter
[ [ key? ] curry all? ] curry filter
```

There is a mapping from fried quotations to lexical closures as defined in the **locals** vocabulary. Namely, a fried quotation is equivalent to a `[| | ]` form where each local binding is only used once, and bindings are used in the same order in which they are defined. The following two lines are equivalent:

```
’[ 3 _ + 4 _ / ]
[| a b | 3 a + 4 b / ]
```

Objects

An *object* is any datum which may be identified. All values are objects in Factor. Each object carries type information, and types are checked at runtime; Factor is dynamically typed.

Equality
Linear order protocol
Classes
Tuples
Generic words and methods

Advanced features:

Delegation
Mirrors
Low-level slot operations

Equality

There are two distinct notions of "sameness" when it comes to objects.

You can test if two references point to the same object (*identity comparison*). This is rarely used; it is mostly useful with large, mutable objects where the object identity matters but the value is transient:

eq? (obj1 obj2 -- ?)

You can test if two objects are equal in a domain-specific sense, usually by being instances of the same class, and having equal slot values (*value comparison*):

= (obj1 obj2 -- ?)

A third form of equality is provided by **number=**. It compares numeric value while disregarding types.

Custom value comparison methods for use with **=** can be defined on a generic word:

equal? (obj1 obj2 -- ?)

Utility class:

identity-tuple

An object can be cloned; the clone has distinct identity but equal value:

clone (obj -- cloned)

Linear order protocol

Some classes define an intrinsic order amongst instances. This includes numbers, sequences (in particular, strings), and words.

<=> (obj1 obj2 -- <=>)

>=< (obj1 obj2 -- >=<)

compare (obj1 obj2 quot -- <=>)

invert-comparison (<=> -- >=<)

The above words output order specifiers.

Ordering specifiers

Utilities for comparing objects:

after? (obj1 obj2 -- ?)

before? (obj1 obj2 -- ?)

after=? (obj1 obj2 -- ?)

before=? (obj1 obj2 -- ?)

Minimum, maximum, clamping:

min (obj1 obj2 -- obj)

max (obj1 obj2 -- obj)

clamp (x min max -- y)

Out of the above generic words, it suffices to implement **<=>** alone. The others may be provided as an optimization.

Linear order example

See also

Sorting sequences

Classes

Conceptually, a `class` is a set of objects whose members can be identified with a predicate, and on which generic words can specialize methods. Classes are organized into a general partial order, and an object may be an instance of more than one class.

At the implementation level, a class is a word with certain word properties set.

Words for working with classes are found in the **classes** vocabulary.

Classes themselves form a class:

class? (object -- ?)

You can ask an object for its class:

class-of (object -- class)

Testing if an object is an instance of a class:

instance? (object class -- ?)

You can ask a class for its superclass:

superclass (class -- super)

superclasses (class -- supers)

subclass-of? (class superclass -- ?)

Class predicates can be used to test instances directly:

Class predicate words

There is a universal class which all objects are an instance of, and an empty class with no instances:

object

null

Obtaining a list of all defined classes:

classes (-- seq)

There are several sorts of classes:

Built-in classes
Union classes
Intersection classes
Mixin classes
Predicate classes
Singleton classes

Tuples are documented in their own section.

Classes can be inspected and operated upon:

Class operations
Class linearization

See also

Class index

Tuples

Tuples are user-defined classes composed of named slots. They are the central data type of Factor's object system.

Tuple examples

A parsing word defines tuple classes:

TUPLE:

For each tuple class, several words are defined, the class word, a class predicate, and accessor words for each slot.

The class word is used for defining methods on the tuple class; it has the same name as the tuple class. The predicate is named name?. Initially, no specific words are defined for constructing new instances of the tuple. Constructors must be defined explicitly, and tuple slots are accessed via automatically-generated accessor words.

Slot accessors
Tuple constructors
Tuple subclassing
Tuple slot declarations
Protocol slots
Tuple introspection

Tuple classes can be redefined; this updates existing instances:

Tuple redefinition

Tuple literal syntax is documented in **Tuple syntax**.

Generic words and methods

A *generic word* is composed of zero or more *methods* together with a *method combination*. A method *specializes* on a class; when a generic word is executed, the method combination chooses the most appropriate method and calls its definition.

A generic word behaves roughly like a long series of class predicate conditionals in a **cond** form, however methods can be defined in independent source files, reducing coupling and increasing extensibility. The method combination determines which object the generic word will *dispatch* on; this could be the top of the stack, or some other value.

Generic words which dispatch on the object at the top of the stack:

GENERIC:

A method combination which dispatches on a specified stack position:

GENERIC#

A method combination which dispatches on the value of a variable at the time the generic word is called:

HOOK:

A method combination which dispatches on a pair of stack values, which must be numbers, and upgrades both to the same type of number:

MATH:

Method definition:

M:

Generic words must declare their stack effect in order to compile. See **Stack effect declarations**.

Method precedence
Calling less-specific methods
Custom method combination
Generic word introspection

Generic words specialize behavior based on the class of an object; sometimes behavior needs to be specialized on the object's *structure*; this is known as *pattern matching* and is implemented in the **match** vocabulary.

Delegation

The**delegate** vocabulary implements run-time consultation for method dispatch.

A *protocol* is a collection of related generic words. An object is said to *consult* another object if it implements a protocol by forwarding all methods onto the other object.

Using this vocabulary, protocols can be defined and consultation can be set up without any repetitive boilerplate.

Unlike **Tuple subclassing**, which expresses *is-a* relationships by statically including the methods and slots of the superclass in all subclasses, consultation forwards generic word calls to another distinct object.

Defining new protocols:

PROTOCOL:

define-protocol (protocol wordlist --)

Defining new protocols consisting of slot accessors:

SLOT-PROTOCOL:

Defining consultation:
BROADCAST:

CONSULT:

define-consult (consultation --)

Every tuple class has an associated protocol consisting of all of its slot accessor methods. The**delegate.protocols** vocabulary defines formal protocols for the various informal protocols used in the Factor core, such as **Sequence protocol**, **Associative mapping protocol** or **Stream protocol**

Mirrors

The**mirrors** vocabulary defines data types which present an object's slots and slot values as an associative structure. This enables idioms such as iteration over all slots in a tuple, or editing of tuples, sequences and assocs in a generic fashion. This functionality is used by developer tools and meta-programming utilities.

A mirror provides such a view of a tuple:
mirror

<mirror> (object -- mirror)

Utility word used by developer tools which inspect objects:
make-mirror (obj -- assoc)

See also

Low-level slot operations

Low-level slot operations

The**slots** vocabulary contains words for introspecting the slots of an object. A *slot* is a component of an object which can store a value.

Tuples are composed entirely of slots, and instances of **Built-in classes** consist of slots together with intrinsic data.

The`"slots"` word property of built-in and tuple classes holds an array of *slot specifiers* describing the slot layout of each instance.
slot-spec

The four words associated with a slot can be looked up in the **accessors** vocabulary:
reader-word (name -- word)

writer-word (name -- word)

setter-word (name -- word)

changer-word (name -- word)

Looking up a slot by name:
slot-named (name specs -- spec/f)

Defining slots dynamically:

define-reader (class slot-spec --)

define-writer (class slot-spec --)

define-setter (name --)

define-changer (name --)

define-slot-methods (class slot-spec --)

define-accessors (class specs --)

Unsafe slot access:

slot (obj m -- value)

set-slot (value obj n --)

See also

Slot accessors, **Mirrors**

Exception handling

Support for handling exceptional situations such as bad user input, implementation bugs, and input/output errors is provided by a set of words built using continuations.

Two words raise an error in the innermost error handler for the current dynamic extent:

throw (error -- *)

rethrow (error -- *)

Words for establishing an error handler:

cleanup (try cleanup-always cleanup-error --)

recover (..a try: (..a -- ..b) recovery: (..a error -- ..b) -- ..b)

ignore-errors (quot --)

Syntax sugar for defining errors:

ERROR:

Unhandled errors are reported in the listener and can be debugged using various tools. See **The debugger**.

Restartable errors
The debugger
Post-mortem error inspection
Common error handling pitfalls

When Factor encounters a critical error, it calls the following word:

die (--)

Restartable errors

Support for restartable errors is built on top of the basic error handling facility. The following words signals recoverable errors:

throw-restarts (error restarts -- restart)

rethrow-restarts (error restarts -- restart)

A utility word using the above:

throw-continue (error --)

The list of restarts from the most recently-thrown error is stored in a global variable:

restarts

To invoke restarts, use **The debugger**.

The debugger

Caught errors can be logged in human-readable form:

print-error (error --)

try (quot --)

User-defined errors can have customized printed representation by implementing a generic word:

error. (error --)

A number of words facilitate interactive debugging of errors:

:error (--)

:s (--)

:r (--)

:c (--)

:get (variable -- value)

Most types of errors are documented, and the documentation is instantly accessible:

:help (--)

If the error was restartable, a list of restarts is also printed, and a numbered restart can be invoked:

:1 (-- *)

:2 (-- *)

:3 (-- *)

:res (n -- *)

You can read more about error handling in **Exception handling**.

Note that in Factor, the debugger is a tool for printing and inspecting errors, not for walking through code. For the latter, see **UI walker**.

Post-mortem error inspection

The most recently thrown error, together with the continuation at that point, are stored in a pair of global variables:

error

error-continuation

Developer tools for inspecting these values are found in **The debugger**.

Common error handling pitfalls

When used correctly, exception handling can lead to more robust code with less duplication of error handling logic. However, there are some pitfalls to keep in mind.

Anti-pattern #1: Ignoring errors

The**ignore-errors** word should almost never be used. Ignoring errors does not make code more robust and in fact makes it much harder to debug if an intermittent error does show up when the code is run under previously unforseen circumstances. Never ignore unexpected errors; always report them to the user.

Anti-pattern #2: Catching errors too early

A less severe form of the previous anti-pattern is code that makes overly zealous use of

recover. It is almost always a mistake to catch an error, log a message, and keep going. The only exception is network servers and other long-running processes that must remain running even if individual tasks fail. In these cases, place the **recover** as high up in the call stack as possible.

In most other cases, **cleanup** should be used instead to handle an error and rethrow it automatically.

Anti-pattern #3: Dropping and rethrowing

Do not use **recover** to handle an error by dropping it and throwing a new error. By losing the original error message, you signal to the user that something failed without leaving any indication of what actually went wrong. Either wrap the error in a new error containing additional information, or rethrow the original error. A more subtle form of this is using **throw** instead of **rethrow**. The **throw** word should only be used when throwing new errors, and never when rethrowing errors that have been caught.

Anti-pattern #4: Logging and rethrowing

If you are going to rethrow an error, do not log a message. If you do so, the user will see two log messages for the same error, which will clutter logs without adding any useful information.

Deterministic resource disposal

Operating system resources such as streams, memory mapped files, and so on are not managed by Factor's garbage collector and must be released when you are done with them. Failing to release a resource can lead to reduced performance and instability.

Using destructors
Writing new destructors
Resource disposal anti-patterns

See also

Destructor tools

Using destructors

Disposing of an object:

dispose (disposable --)

Utility word for scoped disposal:

with-disposal (object quot --)

Utility word for disposing multiple objects:

dispose-each (seq --)

Utility words for more complex disposal patterns:

with-destructors (quot --)

&dispose (disposable -- disposable)

|dispose (disposable -- disposable)

Writing new destructors

Superclass for disposable objects:

disposable

Parametrized constructor for disposable objects:

new-disposable (class -- disposable)

Generic disposal word:

dispose* (disposable --)

Global set of disposable objects:

disposables

Resource disposal anti-patterns

Words which create objects corresponding to external resources should always be used with **with-disposal**. The following code is wrong:

```
<external-resource> ... do stuff ... dispose
```

The reason being that if `do stuff` throws an error, the resource will not be disposed of. The most important case where this can occur is with I/O streams, and the correct solution is to always use **with-input-stream** and **with-output-stream**; see **Default input and output streams** for details.

Memoization

The **memoize** vocabulary implements a simple form of memoization, which is when a word caches results for every unique set of inputs that is supplied. Calling a memoized word with the same inputs more than once does not recalculate anything.

Memoization is useful in situations where the set of possible inputs is small, but the results are expensive to compute and should be cached. Memoized words should not have any side effects.

Defining a memoized word at parse time:

MEMO:

Defining a memoized word at run time:

define-memoized (word quot effect --)

Clearing memoized results:

reset-memoized (word --)

Parsing words

The Factor parser follows a simple recursive-descent design. The parser reads successive tokens from the input; if the token identifies a number or an ordinary word, it is added to an accumulator vector. Otherwise if the token identifies a parsing word, the parsing word is executed immediately.

Parsing words are defined using the defining word:

SYNTAX:

Parsing words have uppercase names by convention. Here is the simplest possible parsing word; it prints a greeting at parse time:

```
SYNTAX: HELLO "Hello world" print ;
```

Parsing words must not pop or push items from the stack; however, they are permitted to access the accumulator vector supplied by the parser at the top of the stack. That is, parsing words must have stack effect `( accum -- accum )`, where `accum` is the accumulator vector supplied by the parser.

Parsing words can read input, add word definitions to the dictionary, and do anything an ordinary word can.

Because of the stack restriction, parsing words cannot pass data to other words by leaving values on the stack; instead, use **suffix!** to add the data to the parse tree so that it can be evaluated later.

Parsing words cannot be called from the same source file where they are defined, because new definitions are only compiled at the end of the source file. An attempt to use a parsing word in its own source file raises an error:

staging-violation (word -- *)

Tools for implementing parsing words:

Reading ahead
Nested structure
Defining words
Parsing raw tokens
Reflection support for vocabulary search path

Reading ahead

Parsing words can consume input from the input stream. Words come in two flavors: words that throw upon finding end of file, and words that return **f** upon the same.

Parsing words that throw on end of file:

scan-token (-- str)

scan-word-name (-- string)

scan-word (-- word)

scan-datum (-- word/number)

scan-number (-- number)

scan-object (-- object)

Parsing words that return **f** on end of file:
(scan-token) (-- str/f)

(scan-datum) (-- word/number/f)

A simple example is the **** word:
```
USING: kernel parser sequences ;

IN: syntax

SYNTAX: \ scan-word <wrapper> suffix! ;
```

Nested structure
Recall that the parser loop calls parsing words with an accumulator vector on the stack. The parser loop can be invoked recursively with a new, empty accumulator; the result can then be added to the original accumulator. This is how parsing words for object literals are implemented; object literals can nest arbitrarily deep.

A simple example is the parsing word that reads a quotation:
```
USING: parser sequences ;

IN: syntax

SYNTAX: [ parse-quotation suffix! ;
```

This word uses a utility word which recursively invokes the parser, reading objects into a new accumulator until an occurrence of **]**:
parse-literal (accum end quot -- accum)

There is another, lower-level word for reading nested structure, which is also useful when called directly:
parse-until (end -- vec)

Words such as **]** use a declaration which causes them to throw an error when an unpaired occurrence is encountered:
delimiter

See also
{, **H{**, **V{**, **W{**, **T{**, **}**

Defining words
Defining words add definitions to the dictionary without modifying the parse tree. The simplest example is the **SYMBOL:** word.
```
USING: parser words.symbol ;

IN: syntax

SYNTAX: SYMBOL: scan-new-word define-symbol ;
```

The key factor in the definition of **SYMBOL:** is **scan-new**, which reads a token from the input and creates a word with that name. This word is then passed to **define-symbol**.
scan-new (-- word)

scan-new-word (-- word)

Colon definitions are defined in a more elaborate way:
:

The **:** word first calls **scan-new**, and then reads input until reaching **;** using a utility word:
parse-definition (-- quot)

The **;** word is just a delimiter; an unpaired occurrence throws a parse error:
```
IN: syntax

DEFER: ; ( -- * ) delimiter
```

There are additional parsing words whose syntax is delimited by **;**, and they are all implemented by calling **parse-definition**.

Parsing raw tokens

So far we have seen how to read individual tokens, or read a sequence of parsed objects until a delimiter. It is also possible to read raw tokens from the input and perform custom processing.

One example is the **USING:** parsing word.
```
USING: lexer vocabs.parser ;

IN: syntax

SYNTAX: USING: ";" [ use-vocab ] each-token ;
```

It reads a list of vocabularies terminated by **;**. However, the vocabulary names do not name words, except by coincidence; so **parse-until** cannot be used here. Instead, a set of lower-level combinators can be used:
each-token (... end quot: (... token -- ...) -- ...)

map-tokens (... end quot: (... token -- ... elt) -- ... seq)

parse-tokens (end -- seq)

Reflection support for vocabulary search path

The parsing words described in **Syntax to control word lookup** are implemented using the below words, which you can also call from your own parsing words.

The current state used for word search is stored in a *manifest*:
manifest

Words for working with the current manifest:
use-vocab (vocab --)

unuse-vocab (vocab --)

add-qualified (vocab prefix --)

add-words-from (vocab words --)

add-words-excluding (vocab words --)

Words used to implement **IN:**:

current-vocab (-- vocab)

set-current-vocab (name --)

Words used to implement **Private words**:

begin-private (--)

end-private (--)

Macros

The**macros** vocabulary implements *macros*, which are code transformations that may run at compile-time under the right circumstances.

Macros can be used to implement combinators whose stack effects depend on an input parameter. Since macros are expanded at compile time, this permits the compiler to infer a static stack effect for the word calling the macro.

Macros can also be used to calculate lookup tables and generate code at compile time, which can improve performance, raise the level of abstraction, and simplify code.

Factor macros are similar to Lisp macros; they are not like C preprocessor macros.

Defining new macros:

MACRO:

A slightly lower-level facility, *compiler transforms*, allows an ordinary word definition to co-exist with a version that performs compile-time expansion. The ordinary definition is only used from code compiled with the non-optimizing compiler. Under normal circumstances, macros should be used instead of compiler transforms; compiler transforms are only used for words such as **cond** which are frequently invoked during the bootstrap process, and this having a performant non-optimized definition which does not generate code on the fly is important.

define-transform (word quot n --)

See also

Generalized shuffle words and combinators, **Fried quotations**

Continuations

At any point in the execution of a program, the *current continuation* represents the future of the computation.

Words for working with continuations are found in the **continuations** vocabulary; implementation details are in **continuations.private**.

Continuations can be reified with the following two words:

callcc0 (quot --)

callcc1 (quot -- obj)

Another two words resume continuations:

continue (continuation -- *)

continue-with (obj continuation -- *)

Continuations as control-flow:

attempt-all (... seq quot: (... elt -- ... obj) -- ... obj)

with-return (quot --)

Continuations serve as the building block for a number of higher-level abstractions, such as **Exception handling** and **Co-operative threads**.

Continuation implementation details

Continuation implementation details

A continuation is simply a tuple holding the contents of the five stacks:

continuation

>continuation< (continuation -- data call retain name catch)

The five stacks can be read and written:

datastack (-- array)

set-datastack (array --)

retainstack (-- array)

set-retainstack (array --)

callstack (-- callstack)

set-callstack (callstack -- *)

namestack (-- namestack)

set-namestack (namestack --)

catchstack (-- catchstack)

set-catchstack (catchstack --)

Vocabulary loader

The**USE:** and **USING:** words load vocabularies using the vocabulary loader. The vocabulary loader is implemented in the **vocabs.loader** vocabulary.

The vocabulary loader searches for vocabularies in a set of directories known as vocabulary roots.

Vocabulary roots

Vocabulary names map directly to source files inside these roots. A vocabulary named `foo.bar` is defined in `foo/bar/bar.factor`; that is, a source file named `bar.factor` within a `bar` directory nested inside a `foo` directory of a vocabulary root. Any level of nesting, separated by dots, is permitted.

The vocabulary directory - `bar` in our example - contains a source file:

- `foo/bar/bar.factor` - the source file must define words in the `foo.bar` vocabulary with an `IN: foo.bar` form

Two other Factor source files, storing documentation and tests, respectively, may optionally be placed alongside the source file:

- `foo/bar/bar-docs.factor` - documentation, see **Writing documentation**
- `foo/bar/bar-tests.factor` - unit tests, see **Unit testing**

Optional text files may contain metadata.

Vocabulary metadata

Vocabulary icons

Vocabularies can also be loaded at run time, without altering the vocabulary search path. This is done by calling a word which loads a vocabulary if it is not in the image, doing nothing if it is:

require (object --)

The above word will only ever load a vocabulary once in a given session. Sometimes, two vocabularies require special code to interact. The following word is used to load one vocabulary when another is present:

require-when (if then --)

There is another word which unconditionally loads vocabulary from disk, regardless of whether or not is has already been loaded:

reload (name --)

For interactive development in the listener, calling **reload** directly is usually not necessary, since a better facility exists for **Runtime code reloading**.

Application vocabularies can define a main entry point, giving the user a convenient way to run the application:

MAIN:

run (vocab --)

runnable-vocab

See also

Vocabularies, **The parser**, **Source files**

Vocabulary roots

The vocabulary loader searches for vocabularies in one of the root directories:

vocab-roots

The default set of roots includes the following directories in the Factor source directory:

`core` - essential system vocabularies such as **parser** and **sequences**. The vocabularies in this root constitute the boot image; see **Bootstrapping new images**.

`basis` - useful libraries and tools, such as **compiler**, **ui**, **calendar**, and so on.

`extra` - additional contributed libraries.

`work` - a root for vocabularies which are not intended to be contributed back to Factor.

You can store your own vocabularies in the `work` directory.

Working with code outside of the Factor source tree

Vocabulary metadata

Vocabulary directories can contain text files with metadata:

`authors.txt` - a series of lines, with one author name per line. These are listed under **Vocabulary authors**.

`platforms.txt` - a series of lines, with one operating system name per line.

`resources.txt` - a series of lines, with one file glob pattern per line. Files inside the vocabulary directory whose names match any of these glob patterns will be included with the compiled application as **Deployed resource files**.

`summary.txt` - a one-line description.

`tags.txt` - a series of lines, with one tag per line. Tags help classify the vocabulary. Consult **Vocabulary tags** for a list of existing tags you can reuse.

Words for reading and writing `summary.txt`:

vocab-summary (vocab -- summary)

set-vocab-summary (string vocab --)

Words for reading and writing `authors.txt`:

vocab-authors (vocab -- authors)

set-vocab-authors (authors vocab --)

Words for reading and writing `tags.txt`:

vocab-tags (vocab -- tags)

set-vocab-tags (tags vocab --)

add-vocab-tags (tags vocab --)

Words for reading and writing `platforms.txt`:

vocab-platforms (vocab -- platforms)

set-vocab-platforms (platforms vocab --)

Words for reading and writing `resources.txt`:

vocab-resources (vocab -- patterns)

set-vocab-resources (patterns vocab --)

Getting and setting arbitrary vocabulary metadata:

vocab-file-contents (vocab name -- seq)

set-vocab-file-contents (seq vocab name --)

Vocabulary icons

An icon file representing the vocabulary can be provided for use by **Application deployment**. If any of the following files exist inside the vocabulary directory, they will be used as icons when the application is deployed.

- `icon.ico` on Windows
- `icon.icns` on MacOS X
- `icon.png` on Linux and *BSD